```
//================================================================
//
//  Draft C++ Interface for the FE Sparse Linear Solver Abstraction
//
//  Distributed Systems Research Dept.
//  Sandia National Labs
//  Livermore, CA 94550
//
//  Authors: r. l. clay - snl
//           k.d. mish - csu chico
//           ivan otero - llnl
//           lee taylor - snl
//           alan williams - snl
//
//  Date:    First OO draft 10-feb-97 (version 0), based on procedural
//           draft started summer 96.
//
//  Update:  15-Oct-98, version 1.0
//
//
//  This interface specification is intended to provide a simple means for
//  passing finite-element data to and obtaining solution services from
//  solver libraries within an object-oriented, C++ setting.
//
//  A procedural interface specification is also in progress, with basic
//  functionality mirroring this C++ spec.
//
//  Updates to this specification can be downloaded from:
//  http://www.ca.sandia.gov/isis/fei_docs.html
//
//================================================================
//
//  Summary of modifications from last updates:
//
//  15-oct-98  added a raft of modifications for v1.0 -- see the
//             annotated reference document for these details.  These
//             modifications include permitting different-sized
//             worksets between the init and load steps, and a list
//             of modifications pertaining to adding "field" abstractions
//             to simplify handling multiphysics simulations.  In v1.0
//             and future versions, all b.c.'s and constraints are dealt
//             with at the level of nodal field parameters, and element
//             blocks register their approximation fields explicitly.
//
//  15-jan-98  added the "resetSystem" function, allowing the FE
//             application to reset the underlying matrix and vector
//             to contain zeros.
//             also added a parameter to the loadElementData function,
//             with which the application specifies which workset is
//             being loaded.
//
//  29-jul-97  added Sierra-motivated abstraction for
//             global-ID (extended integer) data types to permit better
//             integration with Sierra FE development efforts.  Redefined
//             block to include variable solution cardinality pattern as
//             per suggestions from the Aztec SNLA group.
//
//  16-may-97  augmented data structures for external nodes to handle
//             both send and receive functions.  Generalized handling of
//             penalty constraint conditions.  Added ability to manage
//             groups (sets) of generic constraints.  Simplified the
//             implementation of the Lagrange multiplier constraint form.
//
```

```
//  23-apr-97  fixed order of load process, by placing boundary condition
//             load step before element data loading.  This permits the
//             implementation to modify the sparse matrix for boundary
//             conditions at the element level, which is a -lot- simpler!
//             Note: we're leaving the "element-first" ordering for the
//             initialization stage, and using the "b.c.-first" approach
//             for the loading steps.  If this lack of agreement in the
//             calling sequence is a problem it would be straightforward
//             to fix, though we're not doing that change now...
//
//  17-apr-97  clarity improvements, including appropriate const
//             declarations for all passed parameters, and improved
//             or enhanced solution return functions
//
//  10-apr-97  minor fixes, including some naming changes to enhance
//             consistency, and two new solution upload functions:
//              - addition of proposed new solution upload functions for
//                element solution param and Lagrange multiplier return
//
//  31-mar-97  Major restructuring effort, involving several key ideas:
//              - removal of workset terminology
//              - introduction of start/end method calling architecture
//                for overall block structure of data-handling (see more
//                detailed note below)
//              - careful definition of nodal lists/nodal sets
//
//  27-feb-97  Introduced workSet definitions for both element and nodal
//             data, including appropriate initializations.
//
//             Fleshed out distribution of parameter lists, and especially
//             those for boundary and interface condition data.
//
//             Began task of merging annotations into the interface file,
//             to document each passed parameter and returned values.
//
//  12-feb-97  Added initialization for boundary condition and interface
//             condition constraints, so that underlying matrix/vector
//             objects are fully configurable after intitialization
//             sequence.
//
//             Added initComplete function to indicate that all
//             initialization steps have been completed, and that the
//             matrix/vector "objects" can be formed.
//
//             Made SparseLinearEquations object constructor require
//             no arguments, but internally contain pointers to matrix
//             and vector objects corresponding to virtual Ax=b.  These
//             internal objects may (if needed) be constructed/configured
//             when the initComplete function is invoked, since at that
//             point all structural information is available.
//
//             All functions now return int-valued error status.
//
//             Various function renaming to improve consistency.
//
//================================================================
//
//  Design overview:
//
//  The C++ interface is based on the following primary abstractions:
//
//      Sparse linear system of equations...
//
```

```
//      Matrix      virtual representation of matrix object A
//      Vector      virtual representation of vector objects
//      Equations   for linear case, composed of matrix A, solution
//                  vector x, and RHS vector b for Ax=b
//
//      The end-user is required to construct an equations object, denoted
//      SparseLinearEquations (SLE), which contains internal pointers to
//      virtual matrix and vector objects.  The equations object is the
//      central interface between the equation solver services and the
//      FE analysis codes.
//
//      The initialization sequence provides sufficient information for
//      construction (if needed) of matrix and vector objects from within
//      the SLE object.
//
//      Finite elements...
//
//      field         a solution field defined over all or part of the
//                    solution domain, and interpolated over any or all
//                    of the element blocks defined via the interface
//
//      block         a collection of elements and associated nodes
//                    satisfying the following elemental criteria:
//
//                    (1) all have the same number of associated nodes
//                    (2) all have the same pattern of solution cardinality
//                        (i.e., the number of unknowns per node forms a
//                         consistent pattern over the associated nodes)
//                    (3) all elements are local to the same processor
//
//      elemSet       a subset of a block to permit grouping of elements
//                    within a block for convenience in passing element through
//                    the interface layer, on a more-or-less arbitrary scale
//                    ranging from "one element" up to "all the elements on
//                    this processor".  It is important to note that elemSets
//                    inherit the following implicit restriction:
//
//                    (4) all nodes are either local to this processor,
//                        or shared by this processor.
//
//      nodeSet       a collection of individual nodes, grouped to make
//                    aggregate handling of data convenient.  One example
//                    of a nodeSet would be a list of nodes where a
//                    generic boundary condition is specified.
//
//      constrSet     a collection of generic constraints, where the
//                    specific form of the constraint (i.e., the number
//                    of nodes defining the constraint, and the weights
//                    assigned to each nodal solution parameter) is fixed
//                    for each different list of nodes associated with
//                    constraint
//
//
//      In cases where the generic term "set" is used, it should be clear
//      from context which type is assumed.
//
//      The block is fundamental to storing common parametric data. The
//      elemSet is the basic working unit of this interface, where the size
//      of the elemSet typically reflects cache performance.
//
//      Important conventions regarding elemSets and nodeSets:
//
//      (a) The union over all the element sets of all the elements
```

```
//          contained in the collection of elemSets is the entire
//          list of elements associated with the processor.
//
//      (b) The intersection over all the element sets of all the
//          elements contained in the collection of elemSets is NULL.
//
//      Note: conventions (a) and (b) may be relaxed somewhat in the
//            future, in order to accomodate "shadow elements", which
//            are fictitious local copies of elements from neighboring
//            processors, used to simplify or reduce communications
//            during the assembly process.  If restrictions (a) and (b)
//            -are- relaxed to admit shadow elements, then the union
//            may include some (shadow) elements -not- associated with
//            this processor, and the intersection may not be null, but
//            instead contain duplicate copies of shadow elements.  In
//            any case, the discrepancies from the restrictions (a) and
//            (b) presented above will be confined to specific cases,
//            namely sets of shadow elements...
//
//      (c) The union over all the nodal sets of all the nodes
//          contained in the collection of nodeSets is the list of
//          nodes that are "interesting", namely nodes with either
//          boundary data defined, or those that are shared with
//          another processor.  If a node isn't "interesting" in these
//          senses, then it doesn't need to be identified as belonging
//          to any nodeSet, as the solver can figure out which nodes
//          are active by scanning the element connectivity data.
//
//      (d) The intersection over all the nodal sets of all the
//          nodes contained in the collection of nodeSets does not
//          have to be NULL, as it may include nodal data that has some
//          overlap, such as a node shared between processors that also
//          has a boundary condition associated with it.
//
//=============================================================
//
//  Known problems/issues/questions:
//
//  !   We will eventually add a more general form of constraint
//      relation (actually, "relations", as more than one equation will
//      be developed for each constraint, in general) for some of the
//      new slidesurface constraint forms used in ALE3D.  This extension
//      will leave the rest of the interface functions unchanged, so it
//      does not detract from the existing interface architecture.
//
//  *   What values to use for errStat codes?
//
//  &   We are working to expand equation solution services interface:
//      - eigen analysis
//      - multi-level/multi-grid support
//      - extension to nonlinear eqns.
//
//=============================================================
//
//  Definitions and Assumptions:
//
//  SPMD:
//
//          Single Program, Multiple Data - the parallel programming
//          paradigm where each processor executes the same set of
//          instructions as the others, but operates on local data.
//          All function calls in this interface are assumed to be
//          executing in parallel according to an SPMD architecture.
```

```
//
//  boundary node:
//
//          A node which has a boundary condition associated with it
//          (note that this definition does -not- imply such a node is
//          located on the boundary between processors of a domain
//          decomposition!).
//
//  active node:
//
//          A node which appears in the element connectivity list
//          for a given processor.  If a node is active, then there
//          will be element data contributing to equations associated
//          with that node.  The union over all the elements of the
//          nodes associated with each element is the list of active
//          nodes, and such nodes are instrumental to constructing
//          the sparse matrix's representation.
//
//          also, "active node" can be taken to be the restriction
//          of this definition to a particular block structure on a
//          given processor, so that we can refer to the list of
//          active nodes associated with a given processor, or a
//          similar (subset) list associated with a given block.
//
//  shared node:
//
//          A node which is shared among two or more processors,
//          and which is connected to an element on this processor.
//          Shared nodes will be a subset of the nodes found by
//          scanning all elements on the local processor (i.e., the
//          list of all active nodes).
//
//  external node:
//
//          A node which is external to the local processor, but
//          which is needed for local computations. External nodes
//          are not connected to any local elements. That is,
//          external nodes will not show up by scanning all elements
//          on the local processor. These are typically associated
//          with interface (e.g., slide line) conditions, and they
//          are by definition -not- active nodes.
//
//
//=============================================================
//
//  description of basic calling architecture
//
//  highest-level overview.....................................
//
//    (1) initialization
//          general data handling to determine eqn system structure
//
//    (2) loading of element and nodal data
//          passing element and boundary condition data so as to
//          construct the system of equations
//
//    (3) equation system solution
//          setting and applying solution strategies for solving
//          the system of equations
//
//    (4) return of solution data
//          query solver to determine nodal and elemental solution
//          values
```

```
//
//  lower-level view.................................................
//
//    --- many steps in calling sequence use a blocked structure to
//        aid in iterating over all the collected data
//
//    (1) initialization
//           (a) general initialization calls
//           (b) element initialization block
//                 (i) pass element set initialization data
//           (c) node initialization block
//                 (i) pass nodal set data (e.g., shared nodes)
//           (d) constraint (interface) condition block
//                 (i) pass constraint relation init data
//           (e) notification of end of initialization section
//
//    Notes on initialization methods:
//
//    -- constraint relation data is passed using constrSets, which
//       are generic aggregations of individual constraint relations.
//       these constrSets can be readily degenerated down to
//       individual algebraic constraint relations
//
//    -- element and nodal initialization data is passed using
//       aggregations elemSet and nodeSet, as these datatypes are
//       generic in that they naturally apply to groups of elements
//       or nodes.  These accumulative data types also degenerate
//       gracefully down to the case of "one element" and "one node",
//       in the same manner as for constraint sets
//
//
//    (2) element and nodal data passing processes
//           (a) boundary condition data-passing block
//                 (i) pass boundary condition node set data
//           (b) element data-passing block
//                 (i) pass element set stiffnesses and loads
//           (c) constraint (interface) condition data-passing block
//                 (i) pass constraint relation definitions
//           (d) notification of end of data-passing section
//
//    Notes on data-passing methods:
//
//    -- some repetition of initialization data (such as connectivity
//       data for elements) may be repeated to simplify data caching.
//
//
//    (3) solution
//           (a) select preferred solution methods
//                 (i) advise choice of solver
//                (ii) advise choice of preconditioner
//           (b) set solution control parameters
//                 (i) max iterations
//                (ii) convergence tolerance
//               (iii) etc.
//           (c) invoke solution process
//
//    Notes on solution methods:
//
//    -- may need add other methods here, such as a means to handle
//       divergent iterations or other exception conditions.
//
//    -- have added methods to pass initial vector estimates to
//       the solvers, as the inverses of those "solution return"
```

```
//       methods listed below.
//
//
//    (4) return of solution data
//           (a) block solution query
//                 (i) return solution data for element sets
//                 (ii) return constraint (Lagrange) parameters
//           (b) alternate (future) methods for solution query
//           (c) notification of end of solution query section
//
//    Notes on solution return methods:
//
//    -- we -can- provide extensibility of solution return process,
//       in case of specialized data-passing needs (such as some
//       legacy Fortran codes).
//
//
//=============================================================================
//
//  basic outline of the calling sequence (function names only, with
//  returned error status codes removed for clarity, and with the
/// various utility functions removed in the interest of simplicity)
//
//=============================================================================
//
//  (0) Construction
//  ----------------
//
//      SparseLinearEquations(args);
//
//
//  (1) initialization
//  ------------------
//
//      initSolveStep(args...);
//          initFields(args...);
//          beginInitElemBlock(args...);
//              initElemSet(args...);
//          endInitElemBlock();
//          beginInitNodeSets(args...);
//              initSharedNodeSet(args...);
//              initExtNodeSet(args...);
//          endInitNodeSets();
//          beginInitCREqns(args...);
//              initCRMult(args...);
//              initCRPen(args...);
//          endInitCREqns();
//      initComplete();
//
//
//  (2) load data
//  -------------
//
//      beginLoadNodeSets(args...);
//          loadBCSet(args...);
//      endLoadNodeSets();
//      beginLoadElemBlock();
//          loadElemSet(args...);
//      endLoadElemBlock();
//      beginLoadCREqns(args...);
//          loadCRMult(args...);
//          loadCRPen(args...);
//      endLoadCREqns();
```

```
//      loadComplete();
//
//
//  (3) equation solution
//  ---------------------
//
//      parameters(args...);
//      iterateToSolve(args...);
//
//
//  (4) return of solution data
//  ---------------------------
//
//      getBlockNodeSolution(args...);
//      getBlockFieldNodeSolution(args...);
//      getBlockElemSolution(args...);
//      getCRMultSolution(args...);
//      getCRMultParam(args...);
//
//
//=============================================================================

// include the file where typedefs are for GlobalID, etc

#include "basicTypes.h"
#include "mpi.h"

// public interface definition

class SparseLinearEquations {

  public:

    // construction
    SparseLinearEquations(MPI_Comm FEI_COMM_WORLD, int masterRank=0) {};
    virtual ~SparseLinearEquations() {};

// Structural initialization sequence.............................

    // per-solve-step initialization
    virtual int initSolveStep(int numElemBlocks,
                              int solvType) = 0;

    // identify all the solution fields present in the analysis
    virtual int initFields(int numFields,
                           const int *cardFields,
                           const int *fieldIDs) = 0;

    // begin blocked-element initialization step
    virtual int beginInitElemBlock(GlobalID elemBlockID,
                                   int numNodesPerElement,
                                   const int *numElemFields,
                                   const int *const *elemFieldIDs,
                                   int interleaveStrategy,
                                   int numElemDOF,
                                   int numElemSets,
                                   int numElemTotal) = 0;

    // initialize element sets that make up the blocks
    virtual int initElemSet(int numElems,
                            const GlobalID *elemIDs,
                            const GlobalID *const *elemConn) = 0;
```

```
    // end blocked-element initialization
    virtual int endInitElemBlock() = 0;

    // begin collective node set initialization step.
    virtual int beginInitNodeSets(int numSharedNodeSets,
                                  int numExtNodeSets) = 0;

    // initialize nodal sets for shared nodes
    virtual int initSharedNodeSet(const GlobalID *sharedNodeIDs,
                                  int lenSharedNodeIDs,
                                  const int *const *sharedProcIDs,
                                  const int *lenSharedProcIDs) = 0;

    // initialize nodal sets for external (off-processor) communication
    virtual int initExtNodeSet(const GlobalID *extNodeIDs,
                               int lenExtNodeIDs,
                               const int *const *extProcIDs,
                               const int *lenExtProcIDs) = 0;

    // end node set initialization
    virtual int endInitNodeSets() = 0;

    // begin interface condition set initialization step
    virtual int beginInitCREqns(int numCRMultSets,
                                int numCRPenSets) = 0;

    // interface condition initialization
    // - lagrange multiplier formulation
    virtual int initCRMult(const GlobalID *const *CRNodeTable,
                           const int *CRFieldList,
                           int numMultCRs,
                           int lenCRNodeList,
                           int& CRMultID) = 0;

    // interface condition initialization
    // - penalty function formulation
    virtual int initCRPen(const GlobalID *const *CRNodeTable,
                          const int *CRFieldList,
                          int numPenCRs,
                          int lenCRNodeList,
                          int& CRPenID) = 0;

    // end interface condition list initialization
    virtual int endInitCREqns() = 0;

    // indicate that overall initialization sequence is complete
    virtual int initComplete() = 0;


// FE data load sequence.....................................

    // set a value (usually zeros) througout the linear system
    virtual int resetSystem(double s=0.0) = 0;

    // begin node-set data load step
    virtual int beginLoadNodeSets(int numBCNodeSets) = 0;

    // boundary condition data load step
    virtual int loadBCSet(const GlobalID *BCNodeSet,
                          int lenBCNodeSet,
                          int BCFieldID,
                          const double *const *alphaBCDataTable,
                          const double *const *betaBCDataTable,
```

```
                          const double *const *gammaBCDataTable) = 0;

    // end node-set data load step
    virtual int endLoadNodeSets() = 0;

    // begin blocked-element data loading step
    virtual int beginLoadElemBlock(GlobalID elemBlockID,
                                   int numElemSets,
                                   int numElemTotal) = 0;

    // elemSet-based stiffness/rhs data loading step
    virtual int loadElemSet(int elemSetID,
                            int numElems,
                            const GlobalID *elemIDs,
                            const GlobalID *const *elemConn,
                            const double *const *const *elemStiffness,
                            const double *const *elemLoad,
                            int elemFormat) = 0;

    // end blocked-element data loading step
    virtual int endLoadElemBlock() = 0;

    // begin interface-condition data load step
    virtual int beginLoadCREqns(int numCRMultSets,
                                int numCRPenSets) = 0;

    // lagrange-multiplier interface condition load step
    virtual int loadCRMult(int CRMultID,
                           int numMultCRs,
                           const GlobalID *const *CRNodeTable,
                           const int *CRFieldList,
                           const double *const *CRWeightTable,
                           const double *CRValueList,
                           int lenCRNodeList) = 0;

    // penalty formulation interface condition load step
    virtual int loadCRPen(int CRPenID,
                          int numPenCRs,
                          const GlobalID *const *CRNodeTable,
                          const int *CRFieldList,
                          const double *const *CRWeightTable,
                          const double *CRValueList,
                          const double *penValues,
                          int lenCRNodeList) = 0;

    // end interface-condition data load step
    virtual int endLoadCREqns() = 0;

    // indicate that overall data loading sequence is complete
    virtual int loadComplete() = 0;


// Equation solution services.....................................

    // set parameters associated with solver choice, etc.
    virtual void parameters(int numParams,
                            char **paramStrings) = 0;

    // start iterative solution
    virtual int iterateToSolve() = 0;


// Solution return services.......................................
```

```
    // return all nodal solution params on a block-by-block basis
    virtual int getBlockNodeSolution(GlobalID elemBlockID,
                                     GlobalID *nodeIDList,
                                     int &lenNodeIDList,
                                     int *offset,
                                     double *results) = 0;

    // return nodal solution for one field on a block-by-block basis
    virtual int getBlockFieldNodeSolution(GlobalID elemBlockID,
                                          int fieldID,
                                          GlobalID *nodeIDList,
                                          int& lenNodeIDList,
                                          int *offset,
                                          double *results) = 0;

    // return element solution params on a block-by-block basis
    virtual int getBlockElemSolution(GlobalID elemBlockID,
                                     GlobalID *elemIDList,
                                     int& lenElemIDList,
                                     int *offset,
                                     double *results,
                                     int& numElemDOF) = 0;

    // return Lagrange solution to FE analysis on a constraint-set basis
    virtual int getCRMultParam(int CRMultID,
                               int numMultCRs,
                               double *multValues) = 0;

    // return Lagrange solution to FE analysis on a whole-processor basis
    virtual int getCRMultSolution(int& numCRMultSets,
                                  int *CRMultIDs,
                                  int *offset,
                                  double *results) = 0;


// associated "puts" paralleling the solution return services.
//
// the int sizing parameters are passed for error-checking purposes, so
// that the interface implementation can tell if the passed estimate
// vectors make sense -before- an attempt is made to utilize them as
// initial guesses by unpacking them into the solver's native solution
// vector format (these parameters include lenNodeIDList, lenElemIDList,
// numElemDOF, and numMultCRs -- all other passed params are either
// vectors or block/constraint-set IDs)

    // put nodal-based solution guess on a block-by-block basis
    virtual int putBlockNodeSolution(GlobalID elemBlockID,
                                     const GlobalID *nodeIDList,
                                     int lenNodeIDList,
                                     const int *offset,
                                     const double *estimates) = 0;

    // put nodal-based guess for one field on a block-by-block basis
    virtual int putBlockFieldNodeSolution(GlobalID elemBlockID,
                                          int fieldID,
                                          const GlobalID *nodeIDList,
                                          int lenNodeIDList,
                                          const int *offset,
                                          const double *estimates) = 0;

    // put element-based solution guess on a block-by-block basis
    virtual int putBlockElemSolution(GlobalID elemBlockID,
```

```
                                    const GlobalID *elemIDList,
                                    int lenElemIDList,
                                    const int *offset,
                                    const double *estimates,
                                    int numElemDOF) = 0;

    // put Lagrange solution to FE analysis on a constraint-set basis
    virtual int putCRMultParam(int CRMultID,
                               int numMultCRs,
                               const double *multEstimates) = 0;


// utility functions that aid in integrating the FEI calls.............

// support methods for the "gets" and "puts" of the soln services.

    // return info associated with Lagrange multiplier solution
    virtual int getCRMultSizes(int& numCRMultIDs,
                               int& lenResults) = 0;

    // return info associated with blocked nodal solution
    virtual int getBlockNodeIDList(GlobalID elemBlockID,
                                   GlobalID *nodeIDList,
                                   int& lenNodeIDList) = 0;

    // return info associated with blocked element solution
    virtual int getBlockElemIDList(GlobalID elemBlockID,
                                   GlobalID *elemIDList,
                                   int& lenElemIDList) = 0;

// miscellaneous self-explanatory "read-only" utility functions.............

    virtual int getNumSolnParams(GlobalID globalNodeID) const = 0;

    //  return the number of stored element blocks
    virtual int getNumElemBlocks() const = 0;

    //  return the number of active nodes in a given element block
    virtual int getNumBlockActNodes(GlobalID blockID) const = 0;

    //  return the number of active equations in a given element block
    virtual int getNumBlockActEqns(GlobalID blockID) const = 0;

    //  return the number of nodes associated with elements of a
    //  given block ID
    virtual int getNumNodesPerElement(GlobalID blockID) const = 0;

    //  return the number of equations (including element eqns)
    //  associated with elements of a given block ID
    virtual int getNumEqnsPerElement(GlobalID blockID) const = 0;

    //  return the number of elements associated with this blockID
    virtual int getNumBlockElements(GlobalID blockID) const = 0;

    //  return the number of elements eqns for elems w/ this blockID
    virtual int getNumBlockElemEqns(GlobalID blockID) const = 0;

};
```